1. **What is Kafka?**

   Apache Kafka is a open source distributed event streaming platform. Lets divide these terminologies,

   Herer streaming is of two types.

   Creating Real Time Stream: Sending the stream of continuous data from Paytm to Kafka server is known as Creating Real Time Streaming

   For example I am booking the ticket from Paytm. Here I am not only the user booking the ticket from world wise Paytm may receive multiple requests for different users to book the tickets on each minute or second.

   Processing Real Time Stream: The process of continuously listening ang processing the data from Kafka server is known as processing real time stream.

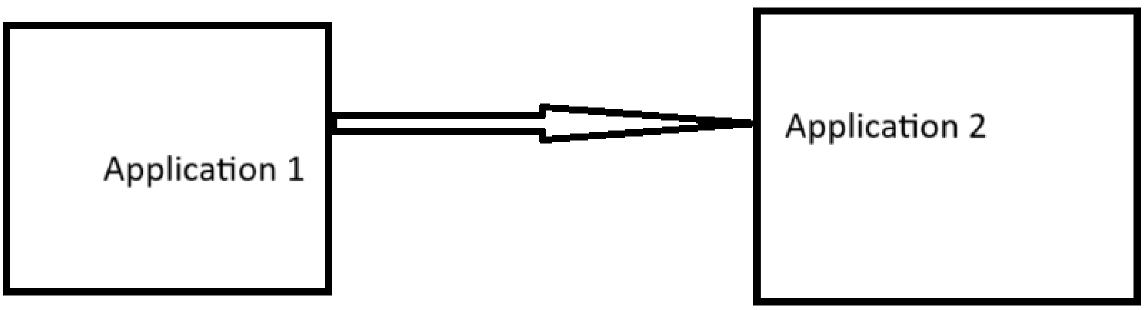
   Distributed: In micro services world everything is load is distributed across the regions to balance the load to avoid the downtime. Likewise Kafka is also distribute Kafka servers in multiple regions to avoid the downtime and load balance. If any of the Kafka server is down another will come into picture and handle the downtime.

2. **Where does Kafka come from?**

   Kafka was originally developed by LinkedIn and it was subsequently open sourced in early 2011 to Apache.

3. **Where do we need Kafka?**

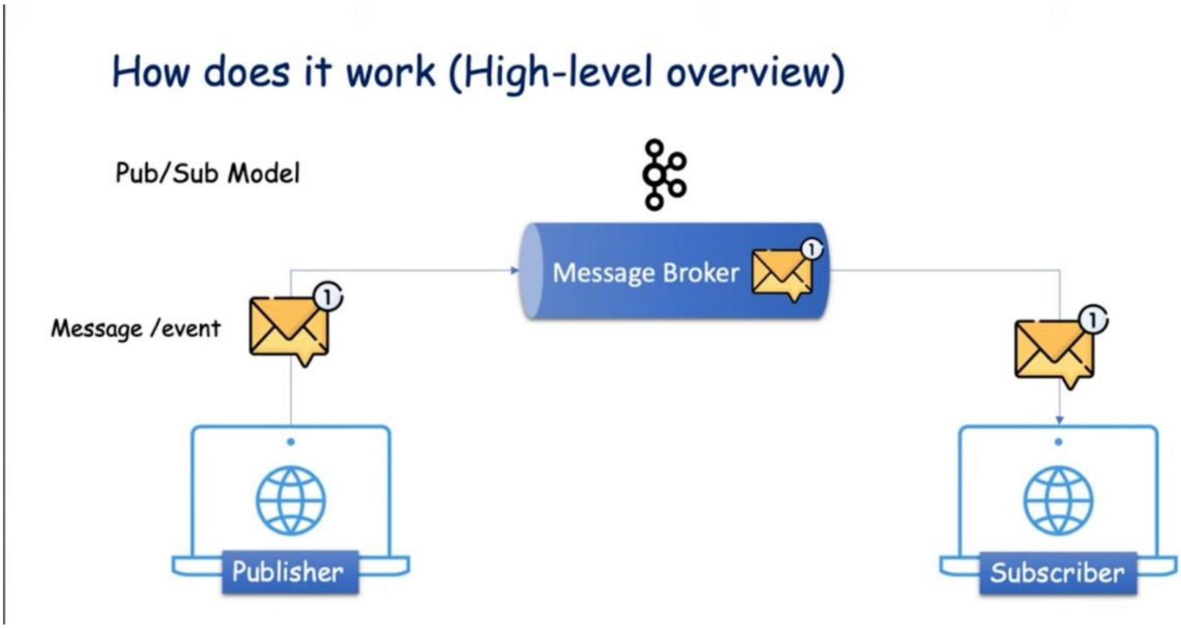
   Consider an example I went to vacation and I got an important parcel from bank. As I was not available at home parcel may went back to provider he tried 3 more attempts to deliver the parcel and I may miss the parcel. In this case if I put the parcel box at my gate postman will put the parcel in and I wont miss the parcel. Here post box will work as middle man between me and postman. Hence post box will be the Kafka.



   In real-time example I am sending the data from application 1 to application 2. In case application 2 is down and not able to receive the data from application 1. Hence data will be lost from Application 1 to Application 2 like postman missed to deliver the letter to me. As I added the post box to receive the letters at my home in case of my unavailability here also I need some middle man to store the data in case of applications is down. Here the middle man is Kafka.

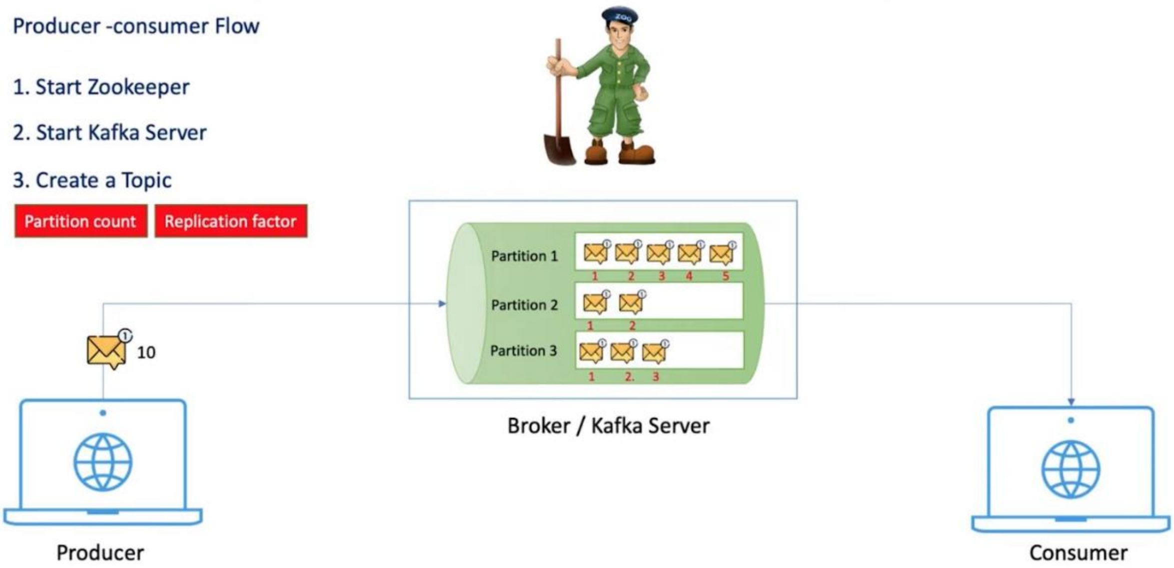
4. How does Kafka work?

Kafka works on pub sub model



Publisher: He is the person who will send the messages.
Message Broker: A broker is nothing but a just server. In simple world a broker is just an intermediate entity that helps in message exchanges between producer and consumer.
Subscriber. Who will consume the messages from Kafka.

Message Broker Internal View



5. **How to start the Kafka in local system?**

    - ✓ Start Zookeeper
    - ✓ Start Kafka Server
    - ✓ Start Producer
    - ✓ Start Consumer

    Zookeeper default port: **2181**
    Kafka Server/Broker default port: **9092**

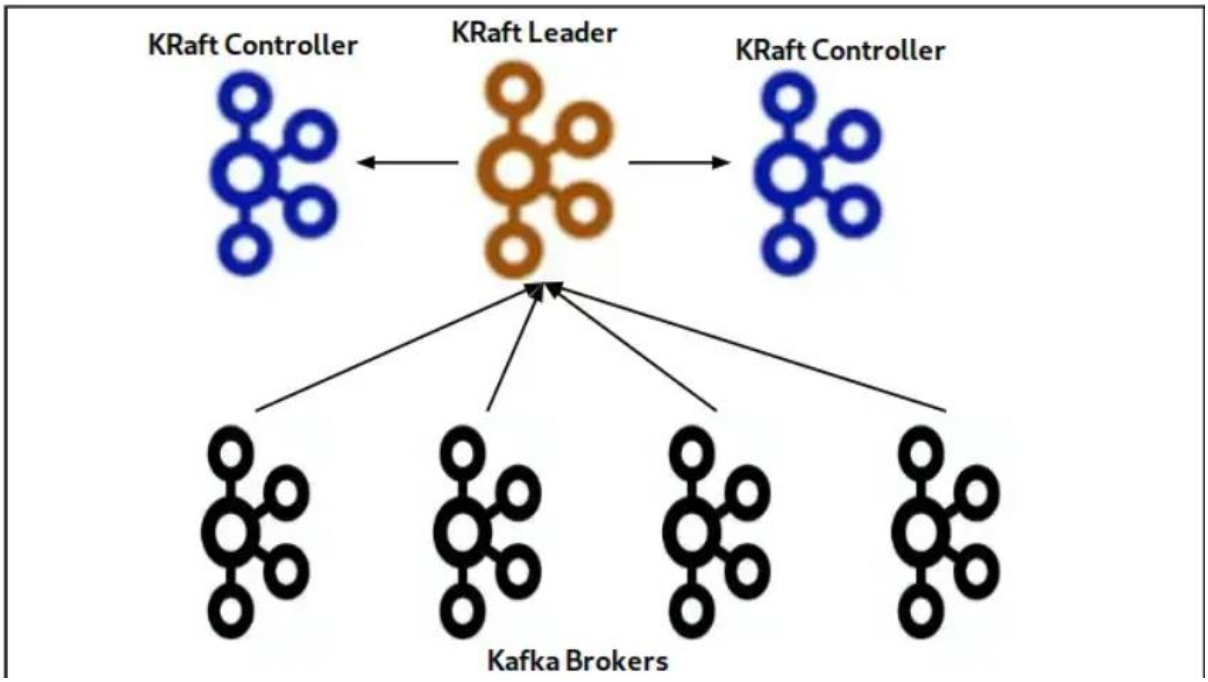
6. **Can we run Kafka with out zookeeper?**

    Apache Kafka has officially deprecated **ZooKeeper** in version 3.5. The introduction of the KRaft mode feature not only enhances the scalability capabilities of Kafka but also simplifies the learning curve by eliminating the need to manage ZooKeeper. This development brings several benefits, making it easier for system administrators to monitor, administer, and support Kafka, while providing developers with a unified security model for the entire system. Additionally, it offers a lightweight single-process deployment option, making it easier to get started with Kafka.

    In the **Kafka Raft Metadata mode**, Kafka stores its metadata and configurations in an internal topic called "__cluster_metadata." This topic consists of a single partition, and the active controller serves as its leader. Managed by an internal quorum, this internal topic is replicated across the cluster. As a result, the cluster nodes can now function as brokers, controllers, or even both (known as combined nodes).
    When the **KRaft mode** is enabled, a few selected servers are designated as controllers, forming the internal quorum. These controllers can operate in either active or standby mode, ready to take over if the currently active controller fails or goes offline.
    With KRaft mode, ZooKeeper nodes are replaced by KRaft Controllers. The process of starting up the controllers involves the following steps:

1. Designation of Kafka Brokers as KRaft controllers.
2. Election of a leader among the controllers.
3. The leader controller contains all previously committed records.
4. Kafka brokers fetch metadata from this KRaft leader.
    In the KRaft mode, every Kafka server now includes an additional configuration parameter called "process.roles." This parameter can be assigned the following values:

- "broker": The Kafka server serves as a broker.
- "controller": The Kafka server acts as a controller of the internal Raft quorum.
- "broker,controller": The Kafka server performs both the roles of a controller and a broker.



In the next section of this tutorial we will see two basic examples of Apache Kafka in KRaft mode. A single node set up and then a cluster set up which we will run on the same machine,

7. **How the leader elections happens in Kafka?**

   Every broker has a information about the list of topics(and partitions) and their leaders which will be kept up to date by the zoo keeper whenever the new leader is elected or when the number of partition changes.
   Thus, when the producer makes a call to one of the brokers, it responds with this information list. Once the producer receives this information, it caches this and uses it to connect to the leader. So next time when it wants to send the message to that particular topic (and partition) it will use this cached information.
   Lets assume there was only one leader and there are no replicas for that topic/ partition duo and it got crushed. In this case it will try to connect to that leader and it fails. It will try to fetch the leader from the other brokers list which it has cached to check if there is any leader for this topic! As it does not find any, it will try to hit to the same leader(that is dead) and after reaching a maximum no of retries it will throw an exception !!

8. **How replication works in kafka?**

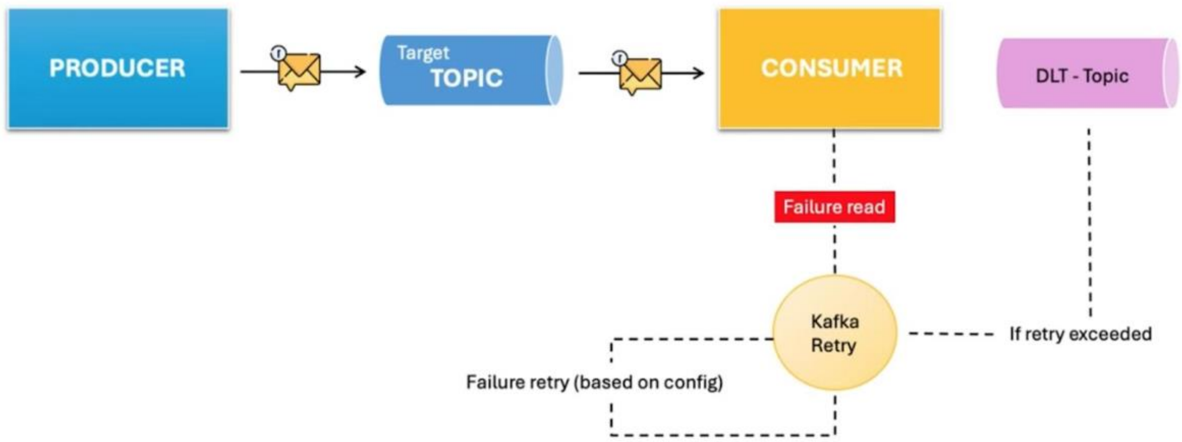
   [Kafka Topic Replication - javatpoint]

   [Replication in Kafka. Replication is the process of having… | by Aman Arora | Medium]

9. **What is the default retry count of Kafka?**

3

10. How to handle failures in Kafka?



Consider a scenario if you are processing the financial transaction failed due to system temporary issue. This can be handled with the help of retry mechanism. It will retry to process the failed transaction based on our configuration. If it is configured for 3 times it will retry for 3 attempts. It will create 2 DLT topics n-1. In case transaction processing failed in 2 attempts   the that transaction will move to DLT (Dead Letter) Topic.

DLT Topic: DLT means Dead Letter Topic. If consumer is unable to process the messages even after retry attempts Kafka will store all the failure transactions events into DLT topic.

References:

Exception Handling in Apache Kafka - GeeksforGeeks
Error Handling Patterns in Kafka

Example: Robust Kafka Consumer Error Handling on a Spring Boot 3 Application | by Noah Hsu | Javarevisited | Medium

11. Why Apache Kafka is so Fast?

Why Apache Kafka is so Fast? - GeeksforGeeks

12. Give brief about Topics, Partitions, and Offsets in Apache Kafka?

1. Topics

- A **Topic** is a logical channel where messages (data) are published.
- Producers send data to a specific topic, and consumers read from it.
- Example: A topic named payment-events could store all payment-related data.

---

2. Partitions

- **Partitions** are sub-divisions of a topic that allow parallelism and scalability.

- Each topic is split into multiple **partitions**, and each partition stores a subset of the topic's data.
- Data within a partition is **ordered**, but there is no global order across partitions.
- Example: If a topic user-logs has 3 partitions, the data will be distributed across these partitions (P0, P1, P2).

Benefits:

- Parallelism: Multiple consumers can read from different partitions simultaneously.
- Scalability: More partitions mean higher throughput.

---

3. Offsets

- An **Offset** is a unique identifier for each message within a partition.
- It is a sequential number assigned to each message as it gets written to a partition.
- Offsets are **immutable** and persist until data retention policies delete them.

Key Concept:

- Consumers use offsets to keep track of their read position.
- Example: If a consumer has processed up to offset 50, it will resume from 51 next time.

Summary Relationship:

- **Topic** = Collection of **Partitions**.
- **Partition** = Ordered sequence of messages identified by **Offsets**.

[Topics, Partitions, and Offsets in Apache Kafka - GeeksforGeeks]

13. Explain possible Kafka failures?

Kafka failures can occur at various levels of the system, affecting the performance and reliability of your distributed messaging setup. Here are some common Kafka failure types and causes:

1. Broker Failures

Crash or Stop: A broker might stop due to resource exhaustion, hardware failure, or misconfiguration.

Network Partitioning: Brokers might lose connectivity, leading to data inconsistency or unavailability.

Disk Failures: Kafka relies heavily on disk storage. If a disk fails, it can result in data loss or unavailability.

2. Producer Failures

Message Loss: If acks=0 or acks=1 is used, messages can be lost before they are fully committed.

Timeouts: Network latency or broker unavailability can cause producer timeouts.

Message Duplication: In case of retries without idempotency enabled, messages might be duplicated.

3. Consumer Failures

Consumer Lag: If consumers fall behind in reading messages, it can result in high lag and delayed processing.

Consumer Group Rebalancing: Frequent rebalances can disrupt message processing.

Offset Management Issues: Incorrect handling of offsets can lead to reprocessing or message loss.

4. Data Loss and Inconsistency

Replication Issues: If replication is misconfigured or insufficient (e.g., replication factor < 3), data loss can occur.

Unclean Leader Elections: If unclean.leader.election.enable is true, data loss can occur when a non-synchronized replica becomes the leader.

5. Zookeeper Failures (for older Kafka versions)

Zookeeper Quorum Loss: If a majority of Zookeeper nodes fail, Kafka might become unavailable.

Leader Election Failure: Zookeeper issues can prevent leader elections from completing.

6. Configuration Errors

Retention Misconfiguration: Data might be deleted too soon if retention periods are too short.

Resource Mismanagement: Insufficient disk space, memory, or CPU can degrade Kafka's performance.

7. Mitigation Strategies

Replication and Acknowledgment: Set acks=all and min.insync.replicas appropriately.

Monitoring and Alerting: Use tools like Prometheus and Grafana for Kafka metrics monitoring.

Fault Tolerance: Ensure proper replication factors and redundancy.

Proper Partitioning: Distribute partitions evenly across brokers. Would you like help diagnosing a specific Kafka issue or configuring failure recovery?

14. Explain about Kafka internal data workflow?

Kafka's internal workflow involves the movement of data from producers to brokers and then to consumers, ensuring fault tolerance, scalability, and high throughput. Here's a step-by-step breakdown of Kafka's internal workflow:

---

1. Producer Workflow (Data Ingestion)

- Producers are client applications that send messages to Kafka.

- Steps:
    - Producer connects to a Kafka broker.
    - Producer sends messages to a topic.
    - If a key is provided, Kafka hashes the key to determine the partition.
    - If no key is provided, Kafka distributes messages in a round-robin fashion across partitions.

✅ Key Concept: Producers can send messages with acks=all for guaranteed durability by waiting for all replicas to acknowledge the message.

---

2. **Partitioning and Storage (Broker Side)**

- Each topic is divided into partitions.
- Each partition is stored across one or more brokers.
- Kafka writes data in an append-only log for each partition.

Leader-Follower Model:

- Each partition has a leader broker responsible for write and read operations.
- Other brokers maintain follower replicas.

✅ Key Concept: Messages are written sequentially to disk, making Kafka extremely fast due to optimized disk usage and the OS page cache.

---

3. Replication and Fault Tolerance

- Kafka ensures fault tolerance through replication.
- Each partition has multiple replicas based on the replication factor.
- Leader Broker: Handles all read and write operations.
- Follower Brokers: Sync data from the leader.

✅ ISR (In-Sync Replicas):

- Brokers that are fully synchronized with the leader.
- If the leader fails, one of the ISR replicas is elected as the new leader.

---

4. Consumer Workflow (Data Consumption)

- Consumers read messages from partitions.
- Consumer Groups:
    - Consumers subscribe to a topic as part of a consumer group.

- Each partition is consumed by only one consumer in the group to avoid duplicate processing.

✅ Key Concept: Consumers pull data from brokers, controlling the rate of consumption.

---

5. Offset Management (Tracking Read Position)

- Offsets represent the position of a message in a partition.
- Kafka stores offsets per consumer group to keep track of consumption progress.
- Offset Storage:
  - Kafka stores offsets in an internal topic called __consumer_offsets.
  - Consumers can use auto-commit or manage offsets manually.

✅ Key Concept: Offsets allow consumers to resume from the last read message after a restart.

---

6. Zookeeper / KRaft (Cluster Management)

- Zookeeper (Older Versions)
  - Manages broker metadata, leader election, and health monitoring.
- KRaft Mode (Kafka 2.8+)
  - Kafka now offers a Zookeeper-free architecture with KRaft for metadata management and leader election.

---

7. Data Retention and Cleanup

- Kafka retains data for a configured time or size, even after being consumed.
- Retention Policies:
  - Time-based: Data is deleted after a set retention period.
  - Size-based: Data is deleted when the partition size exceeds a threshold.
  - Log Compaction: Keeps only the latest message per key for a partition.

✅ **Summary of Kafka Internal Workflow:**

1. Producer sends messages to a Kafka topic.
2. Broker (Leader Partition) receives the message and writes it to disk.
3. Follower Replicas replicate the message for fault tolerance.
4. Consumer reads messages from partitions and commits offsets.

5. Zookeeper/KRaft manages metadata, leader elections, and cluster health.

15. How to handle and recover brokers failure?

Kafka is designed for fault tolerance and high availability. When a broker fails, Kafka can continue functioning without data loss or downtime due to its replication mechanism and leader election process.

1. Broker Failure Scenario:

A Kafka broker failure occurs when a broker goes offline, either due to hardware issues, network failures, or system crashes. Consequences include:

The broker stops handling read/write operations for its partitions.

Partitions handled by the broker may become unavailable temporarily.

2. How Kafka Handles Broker Failures Automatically:

✅ **Step 1: Leader Detection and Failure Identification**

Each partition has a leader broker responsible for reads and writes.

Kafka uses Zookeeper (or KRaft in newer versions) to monitor broker health.

If a broker fails, Zookeeper detects the failure and notifies the cluster.

✅ **Step 2: Leader Election (Failover Mechanism)**

If the leader broker fails, Kafka elects a new leader from the ISR (In-Sync Replicas).

The next available in-sync replica (ISR) becomes the new leader.

If no ISR is available, Kafka may perform an unclean leader election (if enabled).

✅ **Step 3: Client Redirection**

Once a new leader is elected:

Producers and consumers automatically detect the new leader.

Kafka clients reconnect to the new leader broker without manual intervention.

✅ **Step 4: Replication Catch-Up (When Broker Recovers)**

When the failed broker recovers:

It attempts to rejoin the cluster.

Kafka will synchronize its data with the current leader replica.

Once fully synced, it rejoins the ISR.

3. Key Configurations for Better Fault Tolerance:

a. Replication Factor:

Set replication.factor=3 (at least 3 replicas for redundancy).

b. Minimum In-Sync Replicas (min.insync.replicas):

Ensures a minimum number of replicas must acknowledge writes before success.

Recommended: min.insync.replicas=2.

c. Acknowledgment Settings (acks):

Producer setting: acks=all ensures writes are acknowledged only after all replicas sync.

d. Unclean Leader Election (unclean.leader.election.enable):

Disabled by default (false) to avoid data loss.

Enabling it (true) can risk data loss but allows faster recovery if no ISR is available.

4. Manual Broker Recovery Steps (If Automatic Fails):

Identify the Failed Broker:

Use kafka-topics.sh and zookeeper-shell.sh to inspect broker status.

Restart the Broker:

Ensure the broker configuration (server.properties) is correct.

Verify Replicas and ISR:

Check if the broker has rejoined the ISR using:

kafka-topics.sh --describe --topic <topic-name> --bootstrap-server <broker-list>

Reassign Partitions (If Needed):

If partitions remain under-replicated, reassign them using:

kafka-reassign-partitions.sh

5. Best Practices for Handling Broker Failures:

✅ Replication: Use a replication factor ≥ 3.

✅ Monitoring: Use tools like Prometheus, Grafana, or Confluent Control Center for monitoring broker health.

✅ Avoid Unclean Leader Elections: Disable unclean leader election (false).

✅ Automated Recovery: Use Kubernetes or cluster management tools for auto-scaling and healing.